picoUML: Processing class diagrams on small devices

Kai Fabian, Lukas Rögner, Matthias Springer, Claus Steuer
Hasso Plattner Institute
Potsdam, Germany

{kai.fabian, lukas.roegner, matthias.springer, claus.steuer}@student.hpi.uni-potsdam.de

ABSTRACT

We propose a design for editing UML class diagrams on small, touch-enabled devices. As an extension to the UML standard we suggest grouping logically related classes into categories.

Our design consists of two functionally equivalent views. The list representation is a faster way of navigating to classes and associations. In addition, the typical, graphical view allows users to get an overview of the diagram and to layout classes and associations.

INTRODUCTION

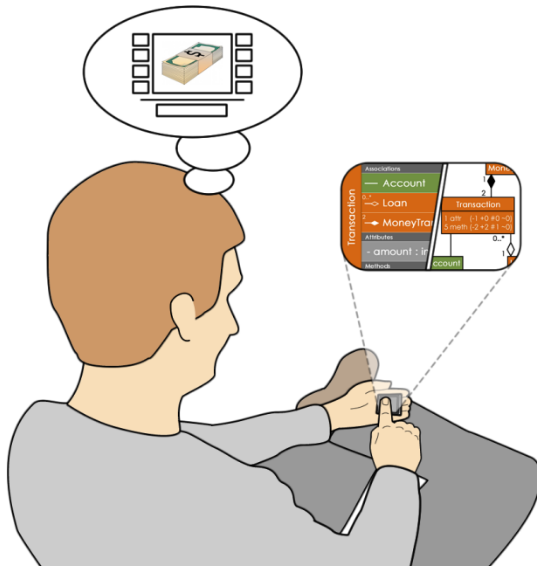

Figure 1: Using picoUML, users can edit existing UML class diagrams or create new diagrams from scratch.

UML class diagrams are used for modeling and documenting a system's structure, such as software systems or domain-specific languages.

Modeling UML class diagrams typically requires large screens. When we asked Stefan Hildebrandt – researcher in domain-specific languages – about his preferred hardware, he told us that he rather "wants to have a second screen". That is why there are many different UML modeling tools available for computers but not for mobile devices.

In this paper we present the UML class editor picoUML, which allows users to create new and modify existing class diagrams.

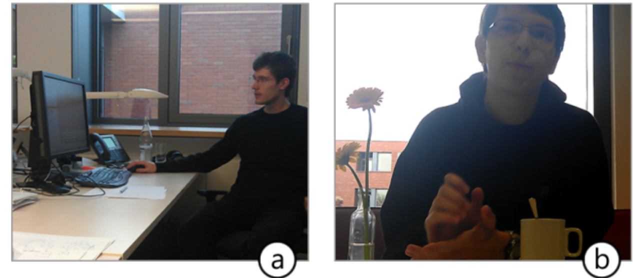

Figure 2: (a) Stefan Hildebrandt creating a domain-specific language for finite automata as a class diagram, (b) Dominik Moritz talking about the use of UML.

DEVICE

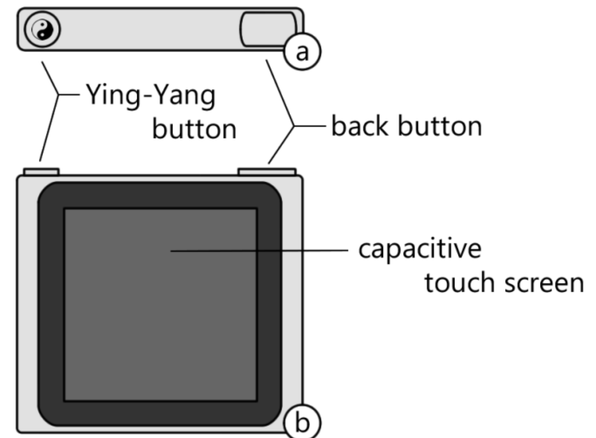

Figure 3: Top (a) and front (b) view of the device.

The target device of the application is a modified version of the iPod nano.

The *Ying-Yang* button allows the user to switch between the graphical view and the list view.

By pressing the *back* button, the user can leave the current dialog at any time. If necessary, he is asked whether he wants to save changes.

WALKTHROUGH

A UML designer wants to edit an existing UML diagram for a bank system. He wants to change the association from *Customer – BankCounter* to *Customer → OnlineBanking* and make it a navigable association on the *OnlineBanking* side.

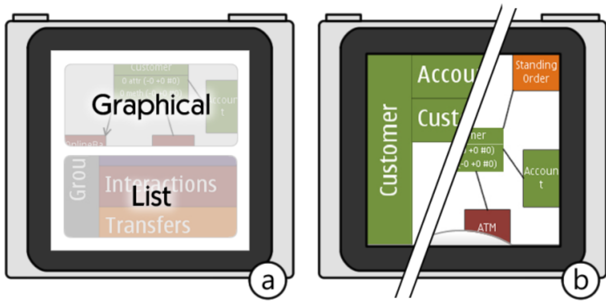

Figure 4: (a) The user chooses between graphical view and list view. (b) By clicking the *Ying-Yang* hardware button he can change between both views.

Both views provide an equivalent way of navigating the diagram. Figure 5 shows the user navigating to the association *Customer–BankCounter* – by using the graphical view (a) and by using the list view (b).

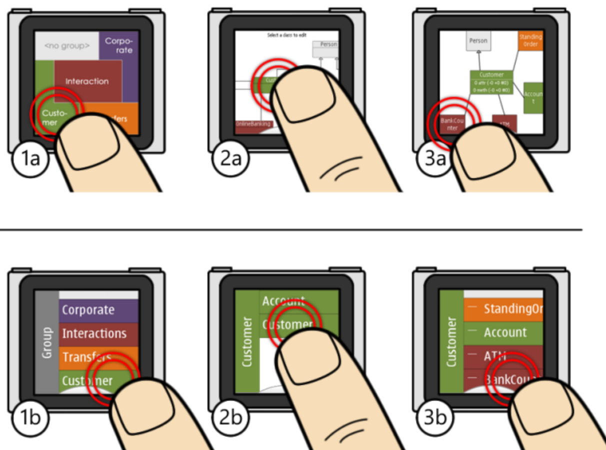

Figure 5: (1) The user selects the category *Customer*, where the class *Customer* is located. (2) Secondly he selects that class. (3) Now he sees an overview of all associations belonging to this class. He taps *BankCounter* to edit the Association *Customer – BankCounter*. (3a) In the graphical view, he selects an association by tapping an association endpoint at the border.

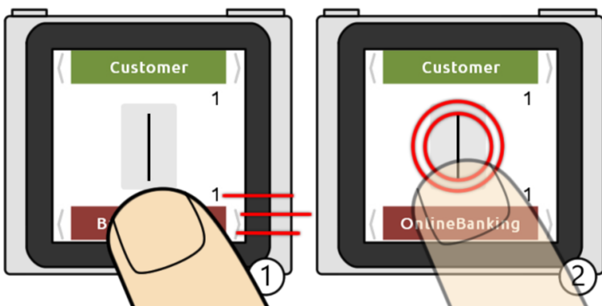

Figure 6: (1) The association details view shows type, targets and multiplicities of the association. By performing a flick gesture on *BankCounter* the user changes the target to *OnlineBanking*. (2) The user taps the association line button to change details of the association.

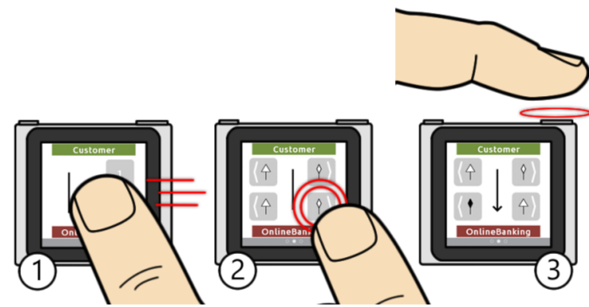

Figure 7: (1) The user doesn't want to change the multiplicities and he flicks left to get to the type changing interface. (2) By tapping the lower right button multiple times, the user changes the type of the association to *Navigable*. (3) The user presses the *back* hardware button to leave the association details view.

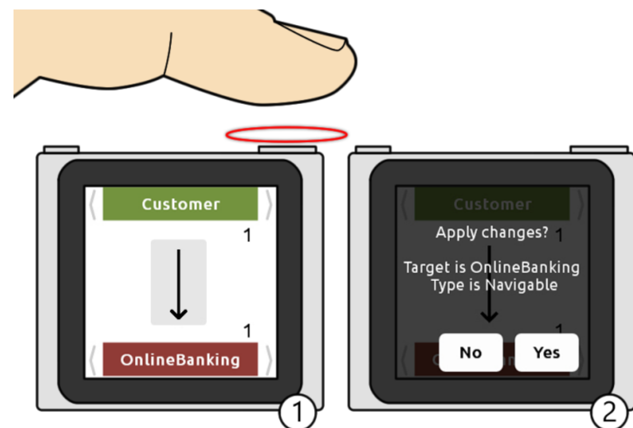

Figure 8: (1) The user presses the *back* hardware button, again, to leave the association details view. (2) He confirms his changes by taping the *Yes* button.

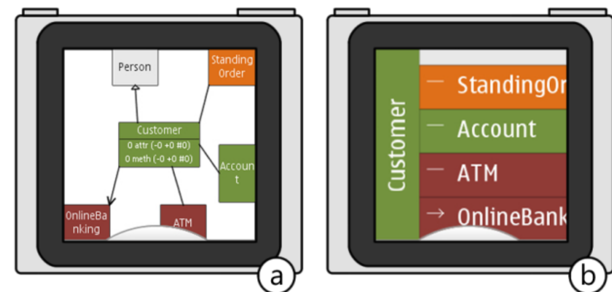

Figure 9: Depending on which view the user started from, he either ends up (a) in the graphical view or (b) in the list view of the class *Customer*. By pressing the *back* button a second time he could switch to the class diagram level (Figure 5/2).

DESIGN

Two functionally equivalent views

During contextual inquiry, Stefan stated that he prefers using a tree-based view for editing class diagrams. He explained the increased efficiency at adding, removing and changing associations with the lack of navigational over-

head in a typically extensive diagram. This conclusion has even more impact on small screen devices, because navigation is already difficult there.

However, Dominik countered that although he could imagine working with a tree-based diagram representation, he still considers a graphical representation important. He explained that a good graphical structure makes a document more readable and understandable. His point of view was also backed up by Stefan’s exemplary walkthrough, when he started by creating and arranging classes within the graphical view, before continuing with refactoring using the tree-based diagram representation.

This concludes our first, most essential design decision of providing two functionally equivalent diagram views. The single functional difference is the absence of graphical arrangement capabilities for classes and associations within the list view. This led to our *first design constraint*: Every single function must be represented within both views in a substantially similar way.

Putting classes into categories

To facilitate document navigation, we introduce a method of structuring class diagrams which is not part of the UML standard: Classes should be partitioned into named, colorized categories. We recommend grouping the classes by logical or functional aspects, although the user is not bound to any constraints when structuring his diagram.

The list view consists of an additional layer expressed by a selective list: In the beginning, the user chooses a category. He can then select the desired class out of a generated list consisting of all classes within the chosen category. When categorizing reasonably, the user will have to choose from a significantly smaller list than before.

For the graphical view we propose a semantic zooming technique, which tries to resemble the local extents of the category’s classes.

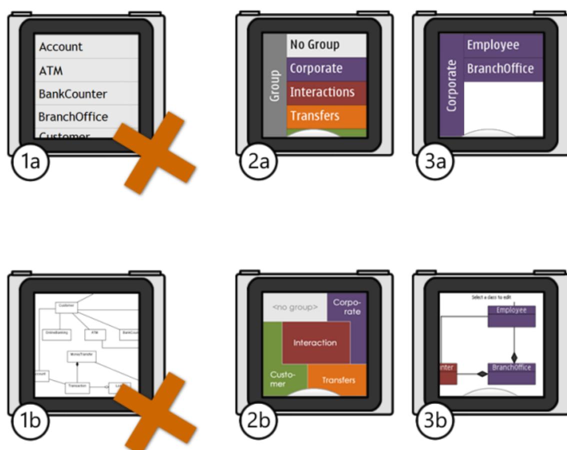

Figure 10: Instead of using a single list (1a) we propose an additional hierarchical level (2a, 3a). Similarly, instead of providing an unlimited zoom level (1b), we propose a semantic zooming technique (2b), which only shows categories. In the graphical view (3b) the user navigates by performing pinching and flicking gestures.

After brainstorming we dismissed our first idea of implementing the list view as a single list of classes. Considering a medium-sized class diagram, such as the bank system diagram in the previous section, such a list would already consist of 15 classes. Although the list could be sorted alphabetically to facilitate finding a specific class, a user would still have to scroll for a distance of four times the screen height.

The same issue arises regarding the graphical view. On large diagrams a user has to scroll many times to move the screen from one border of the diagram to the opposing border. When zooming out, a user can still realize the rough structure of the diagram, if he is familiar with the diagram. However, he can’t determine the exact position of specific classes, since the text becomes unreadable.

Slide-to-descend

In our first design, the list view did not provide any way for going back to a higher hierarchical level, other than the *back* hardware button. In our final design, we provide the user with an additional technique supporting going back, that we call *slide-to-descend*.

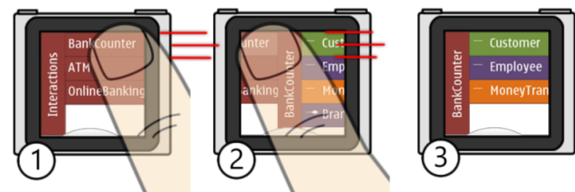

Figure 11: Here, the user does not tap a list element to go deeper within the hierarchy, but slides it sidewards to change the hierarchical level.

By dragging the name of the current class (*BankCounter* in Figure 11) or category to the right, the user can go back without pressing a hardware button. The same technique can be used to go deeper within the hierarchy by dragging an item to the left.

To improve discoverability the user can also tap an item and a sliding animation clarifies the interface’s affordance. During user testing, about two thirds of the testers started using the menu in a sliding manner, after observing the animation. Also, the number of cases in which users traversed to a wrong element decreased – because they can now interrupt the sliding gesture when noticing the wrong selection.

Providing a method to cancel changes

During heuristic evaluation 4 out of 8 evaluators said they either wanted to have a method to cancel changes in association details dialog, or alternatively a method to apply these changes explicitly. Our initial design didn’t provide a *Cancel* or *OK* button and applied changes immediately, when pressing the *back* button.

In our improved design we provide the user with a modal dialog (Figure 8/2), which asks him whether he wants to save changes or not, after he pressed the *back* button. If the

user presses the *back* button a second time, the dialog will vanish and the changes will be canceled.

Spin buttons to change an association's type

In our first design the menu to change the association type consisted of an image of the current association and a button for each end of the association.

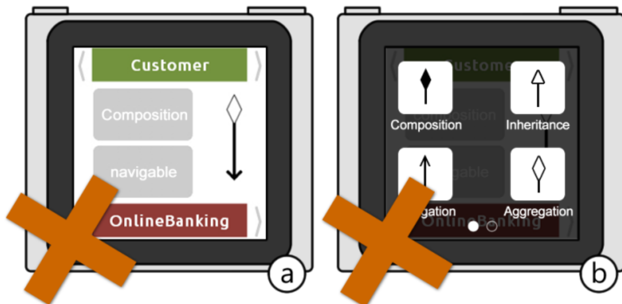

Figure 12: Our initial design, using a modal dialog

Each button had a label indicating the current association type. By pressing a button the user could open a dialog consisting of two pages, which could be scrolled with a flick gesture. Each page contained at most 4 buttons.

During paper prototyping we noticed, that this design consumed a lot of time and that there was no clear mapping between the dialog and the association endpoint being edited. Moreover, about one third of the testers didn't know how to go back without changing the association. To achieve this, they would have had to select the current association type again and therefore memorize the type and the selected association end before entering the menu, or press the *back* hardware button.

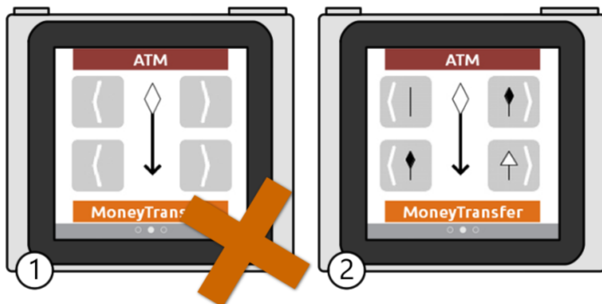

Figure 13: Using spin buttons to change the association type.

In our new design we use spin buttons. The user changes the type of an association by tapping the spin buttons and spinning through the possible types. The new type is immediately shown. Since there are only six different types, this is a fast technique. The missing labels showing the name of an association type are not a problem, because most users refer to a specific type by its visual representation, rather than by its name.

During heuristic evaluation all evaluators reported, that this technique was easy to understand and fast to use. However, some evaluators said, that they would like to see a preview of the next type before pressing the spin button.

Sliding through endpoints in different levels

To edit an association target, the user uses a flick gesture. Doing so, a new class scrolls in, then snaps and thus indicates the new association endpoint. While heuristic evaluation most evaluators reported, that they liked this technique, but that it took too long to scroll to a specific class, especially when diagrams get bigger.

To solve this problem we developed a two-stage selection process, which benefits from our separation of classes into categories. The user starts in the first stage, where he can scroll through all classes of the diagram. The classes are sorted according to their categories, and within a category, sorted alphabetically. By tapping a class, the second stage is entered. The user scrolls through the different categories, starting at the category of the current class. Tapping a category brings the user back to the first stage, now starting at the first class of the selected category.

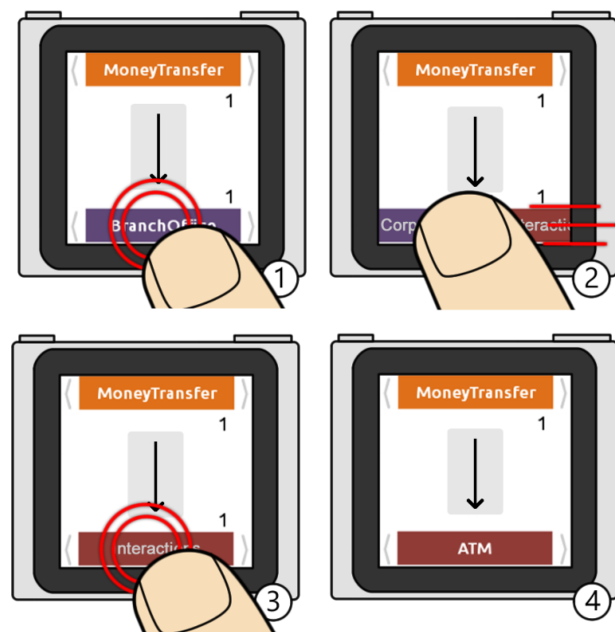

Figure 14: (1) By tapping the class *BranchOffice*, the user reaches the category *Corparate*. (2) The user scrolls to the category *Interactions* and (3) confirms it by tapping. (4) Now he navigates to the class *ATM*.

This way of selecting a target enables users to quickly navigate through the classes in big diagrams. If the diagram is small or the user doesn't know about this feature he can remain in the first stage, without losing the ability to access each class of the diagram.

CONCLUSION

During contextual inquiry we learned that software developers frequently have to make changes to existing UML diagrams.

Therefore we put particular focus on efficiency by providing the slide-to-descent list view and the same hierarchical structure when sliding through association endpoints.

We believe future applications can benefit from our design, especially from the hierarchical list view and our slide-to-descent technique.